A GPU-based implementation for Range Queries on Spaghettis Data Structure

Roberto Uribe-Paredes[1,2], Pedro Valero-Lara[3],
Enrique Árias[4], José L. Sánchez[4], Diego Cazorla[4]

[1] Computer Engineering Department, University of Magallanes, UMAG,
Punta Arenas, Chile.
[2] Database Group - UART, National University of Patagonia Austral,
Río Turbio, Santa Cruz, Argentina.
[3] Albacete Research Institute of Informatics, University of Castilla-La Mancha,
Albacete, España.
[4] Computing Systems Dept, University of Castilla-La Mancha,
Albacete, España.
e-mail: roberto.uribeparedes@gmail.com

Abstract. Similarity search in a large collection of stored objects in a metric database has become a most interesting problem. The *Spaghettis* is an efficient metric data structure to index metric spaces. However, for real applications processing large volumes of generated data, query response times can be high enough. In these cases, it is necessary to apply mechanisms in order to significantly reduce the average query time. In this sense, the parallelization of metric structures is an interesting field of research. The recent appearance of *GPU*s for general purpose computing platforms offers powerful parallel processing capabilities. In this paper we propose a *GPU*-based implementation for *Spaghettis* metric structure. Firstly, we have adapted *Spaghettis* structure to *GPU*-based platform. Afterwards, we have compared both sequential and *GPU*-based implementation to analyse the performance, showing significant improvements in terms of time reduction, obtaining values of speed-up close to 10. **keywords**: Databases, similarity search, metric spaces, algorithms, data structures, parallel processing, GPU, CUDA.

1 Introduction

In the last decade, the search of similar objects in a large collection of stored objects in a metric database has become a most interesting problem. This kind of search can be found in different applications such as voice and image recognition, data mining, plagiarism and many others. A typical query for these applications is the *range search* which consists in obtaining all the objects that are at a definite distance from the consulted object.

1.1 Similarity Search in Metric Spaces

Similarity is modeled in many interesting cases through metric spaces and the search of similar objects through range search or nearest neighbour. A metric

space $(\mathbb{X}, d)$ is a set $\mathbb{X}$ and a distance function $d : \mathbb{X}^2 \rightarrow \mathbb{R}$, so that $\forall x, y, z \in \mathbb{X}$; then there must be properties of positiveness ($d(x, y) \geq 0$ *and* $d(x, y) = 0$) iff ($x = y$), symmetry ($d(x, y) = d(y, x)$) and triangle inequality ($d(x, y) + d(y, z) \geq (d(x, z))$.

In a metric space $(\mathbb{X}, d)$ given, a finite data set $\mathbb{Y} \subseteq \mathbb{X}$, a series of queries can be made. The basic query is the *range query*, a query being $x \in \mathbb{X}$ and a range $r \in \mathbb{R}$. The range query around x with range r is the set of objects $y \in \mathbb{Y}$ such that $d(x, y) \leq r$. A second type of query that can be built using the range query is *k nearest neighbour*, the query being $x \in \mathbb{X}$ and object k. Neighbors k nearest to x are a subset $\mathbb{A}$ of objects $\mathbb{Y}$, such that if $|\mathbb{A}| = k$ and an object $y \in \mathbb{A}$ does not exist an object $z \notin A$ such that $d(z, x) \leq d(y, x)$.

Metric access methods, *metric space indexes* or *metric data structures* are different names for data structures built over a set of objects. The objective of these methods is to minimize the amount of distance evaluations made to solve the query. Searching methods for metric spaces are mainly based on dividing the space using the distance to one or more selected objects. As they do not use particular characteristics of the application, these methods work with any type of objects [1].

Among other important characteristics of metric structures, we can mention that some methods may work only with discrete distances, while others also accept continuous distances. Some methods are static, since the data collection cannot grow once the index has been built. Others accept insertions after construction. Some dynamic methods allow insertions and deletions once the index has been generated.

Metric space data structures can be grouped in two classes [1], *clustering*-based and *pivots*-based methods.

The *clustering*-based structures divide the space into areas, where each area has a so-called center. Some data is stored in each area, which allows easy discarding the whole area by just comparing the query with its center. Algorithms based on clustering are better suited for high-dimensional metric spaces, which is the most difficult problem in practice. Some clustering-based indexes are *BST* [2], *GHT* [3], *M-Tree* [4], *GNAT* [5], *EGNAT* [6], and *SAT* [7].

There exist two criteria to define the areas in clustering-based structures: *hyperplanes* and *covering radius*. The former divides the space in *Voronoi* partitions and determines the hyper plane the query belongs to according to the corresponding center. The covering radius criterion divides the space in spheres that can be intersected and one query can belong to one or more spheres.

The *Voronoi diagram* is defined as the plane subdivision in n areas, one per each center c_i of the set $\{c_1, c_2, \ldots, c_n\}$ (centers) so that a query $q \in c_i$ area if and only if the Euclidean distance $d(q, c_i) < d(q, c_j)$ for every c_j, with $j \neq i$.

In the *pivots*-based methods, a set of pivots are selected and the distances between the pivots and database elements are precalculated. When a query is made, the query distance to the pivots is calculated and the triangle inequality is used to discard the candidates. Its objective is to filter objects during a request

through the use of a triangular inequality, without really measure the distance between the object under request and the discarded object.

An abstract view of this kind of algorithms is the following:

- A set of k pivots ($\{p_1, p_2, \ldots, p_k\} \in \mathbb{X}$) are selected. During indexing time, for each object x from the database $\mathbb{Y}$, the distance to the k pivots is calculated and stored ($d(x, p_1), \ldots, d(x, p_k)$).
- Given a query (q, r), the result $d(p_i, x) \leq d(p_i, q) + d(q, x)$ is obtained by triangular inequality, with $x \in \mathbb{X}$. In the same way, $d(p_i, q) \leq d(p_i, x) + d(q, x)$ is obtained. From these inequations, it is possible to obtain a lower bound for the distance between q and x given by $d(q, x) \geq |d(p_i, x) - d(p_i, q)|$. Thus, the objects x are the objects that accomplish with $d(q, x) \leq r$, and then the rest of objects that do not accomplish with $|d(q, p_i) - d(x, p_i)| \leq r$ can be excluded.

Many indexes are trees, and, the children of each node define areas of space. Range queries traverse the tree, entering into all the children whose areas cannot be proved to be disjoint with the query region. Other metric structures are arrays; in this case, the array usually contains all the objects of the database and maintains the distances to the pivots.

The increased size of databases and the emergence of new types of data, where exact queries are not needed, creates the need to raise new structures to similarity search. Moreover, real applications require that these structures allow them to be stored in secondary memory efficiently, consequently optimized methods for reducing the cost of disk accesses are needed.

Finally, the need to process large volumes of generated data requires to increase processing capacity and so to reduce the average query times. In this context, the study is relevant in terms of parallelization of algorithms and distribution of the database.

1.2 Parallelization of Metric Structures

Currently, there are many parallel platforms for the implementation of metric structures. In this context, basic research has focused on technologies for distributed memory applications, using high level libraries for message passing as MPI [8] or PVM [9], and shared memory, using the language or directives of OpenMP [10].

In [11] and [12] we can find information about testing done on the *MTree*; in this case, the authors focus their efforts on optimizing the structure to properly distribute the nodes on a platform of multiple disks and multiple processors.

Some studies have focused on different structures parallelized on distributed memory platforms using MPI or BSP. In [13] several methods to parallelize the algorithms of construction and search on *EGNAT*, analyzing strategies for distribution of local and/or global data within the cluster, are presented. In [14] the problem of distributing a metric-space search index based on clustering into a set of distributed memory processors, using *List of Clusters* like base structure, is presented.

In terms of shared memory, [15] proposes a strategy to organize metric-space query processing in multi-core search nodes as understood in the context of search engines running on clusters of computers. The strategy is applied in each search node to process all active queries visiting the node as part of their solution which, in general, for each query is computed from the contribution of each search node. Besides, this work proposes mechanisms to address different levels of query traffic on a search engine.

Most of the previous and current works developed in this area are carried out considering classical distributed or shared memory platforms. However, new computing platforms are gaining in significance and popularity within the scientific computing community. Hybrid platforms based on *Graphics Processing Units* (GPU) is an example.

In the present work we show a version of the pivot-based metric structure called *Spaghettis* [16] implemented on a GPU-based platform. There are very little work in metric spaces developed in this kind of platforms. In Section 2.2 we show related work in this area.

2 Graphics Processing Units

The era of single-threaded processor performance increases has come to an end. Programs will only increase in performance if they utilize parallelism. However, there are different kinds of parallelism. For instance, multicore CPUs provide task-level parallelism. On the other hand, Graphics Processing Units (GPUs) provide data-level parallelism.

Current *GPU*s consist of a high number (up to 512 in current devices) of computing cores and high memory bandwidth. Thus, GPUs offer a new opportunity to obtain short execution times. They can offer 10x higher main memory bandwidth and use data parallelism to achieve up to 10x more floating point throughput than the CPUs [17].

GPUs are traditionally used for interactive applications, and are designed to achieve high rasterization performance. However, their characteristics have led to the opportunity to other more general applications to be accelerated in GPU-based platforms. This trend is called General Purpose Computing on GPU (GPGPU) [18]. These general applications must have parallel characteristics and an intense computational load to obtain a good performance.

To assist in the programming tasks of these devices, the GPU manufacturers, like NVIDIA or AMD/ATI, have proposed new languages or even extensions for the most common used high level programming languages. As example, NVIDIA proposes CUDA [19], which is a software platform for massively parallel high-performance computing on the company powerful GPUs.

In CUDA, the calculations are distributed in a mesh or grid of thread blocks, each with the same size (number of threads). These threads run the GPU code, known as kernel. The dimensions of the mesh and thread blocks should be carefully chosen for maximum performance based on the specific problem being treated.

Current GPUs are being used for solving different problems like data mining, robotics, visual inspection, video conferencing, video-on-demand, image databases, data visualization, medical imaging, etc and it is increasingly the number of applications that are being parallelized for GPUs.

2.1 CUDA Programming Model

The *NVIDIA*'s *CUDA* Programming Model ([19]) considers the GPU as a computational device capable to execute a high number of parallel threads. CUDA includes *C/C++* software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware to the developers by means of an Application Programming Interface (API). Among the main tasks to be done in CUDA are the following: allocate data on the GPU, transfer data between the GPU and the CPU and launch kernels.

A CUDA kernel executes a sequential code in a large number of threads in parallel. The threads within a block can work together efficiently exchanging data via a local shared memory and synchronize low-latency execution through synchronization barriers (where threads in a block are suspended until they all reach the synchronization point). By contrast, the threads of different blocks in the same grid can only coordinate their implementation through a high-latency accesses to global memory (the graphic board memory). Within limits, the programmer specifies how many blocks and the number of threads per block that are allocated to the implementation of a given kernel.

2.2 GPUs and Metric Spaces

As far as we know, the solutions considered till now developed on GPUs are based on *kNN* queries without using data structures. This means that GPUs are basically applied to exploit its parallelism only for exhaustive search (brute force) [20–22].

In [20] both elements (A) and queries (B) matrices are divided on fixed size submatrices. In this way, the resultant submatrix C is computed by a block of threads. Once the whole submatrix has been processed, *CUDA-based Radix Sort* [23] is applied over the complete matrix in order to sort it and obtain the first k elements as a final result.

In [21] a brute force algorithm is implemented where each thread computes the distance between an element of a database and a query. Afterwards, it is necessary to sort the resultant array by means of a variant of the *insertion sort* algorithm.

As a conclusion, in these works the parallelization is applied in two stages. The first one consists in building the distance matrix, and the second one consists in sorting this distance matrix in order to obtain the final result.

A particular variant of the above proposed algorithms is presented in [24] where the search is structured into three steps. In the first step each block solves a query. Each thread keeps a heap where stores the kNN nearest elements

Algorithm 1 *Spaghettis*: Construction Algorithm.

1: {Let $\mathbb{X}$ be the metric space}
2: {Let $\mathbb{Y} \subseteq \mathbb{X}$ be the database}
3: {Let P be the set of pivots $p_1, \ldots, p_k \in \mathbb{X}$}
4: {Let S_i be the table of distances associated p_i}
5: {Let Spaghettis be $\cup S_i$}
6: **for all** $p_i \in P$ **do**
7: $S_i \leftarrow d(p_i, \mathbb{Y})$
8: **end for**
9: **for all** S_i **do**
10: Order(S_i)
11: **end for**
12: Each element within S_i stores its position in the next table (S_{i+1})

proccessed by this thread. Secondly, a reduction operation is applied to obtain a final heap. Finally, the first k elements of this final heap are taken as a result of the query.

3 *Spaghettis* Data Structure

Spaghettis [16] is a variant of data structure *LAESA* [25] based on pivots. The method tries to reduce the CPU time needed to carry out a query by using a data structure where the distance to the pivots is sorted independently. As a result there is an array associated to each pivot allowing a binary search in a given range.

For each pivot set $S_i = \{x : |d(x, p_i) - d(q, p_i)| \leq r\}, i = 1, ..., k$, is obtained, where q is a query and r is a range, and a list of candidates will be formed by intersection of the whole sets.

3.1 Construction

During the construction of the spaghettis structure, a random set of pivots $p_1, ..., p_k$ is selected. These pivots could belong or not to the database to be indexed. The algorithm 1 shows in detail the construction process. Each position on table S_i represents an object of the database which has a link to its position on the next table. The last table links the object to its position on the database. Figure 1 shows an example considering 17 elements.

3.2 Searching

During the searching process, given a query q and a range r, a range search on an *spaghettis* follows the following steps:

1. The distance between q and all pivots $p_1, \ldots, p_k$ is calculated in order to obtain k intervals in the form $[a_1, b_1], ..., [a_k, b_k]$, where $a_i = d(p_i, q)$ - r and $b_i = d(p_i, q) + r$.

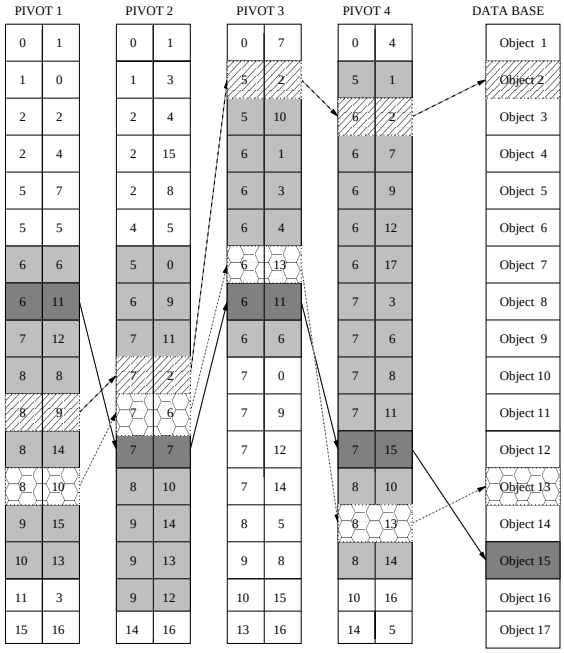


Fig. 1. *Spaghettis*: Construction and search. Example for query q with ranges $\{(6,10),(5,9),(2,6),(4,8)\}$ to pivots.

2. The objects in the intersection of all intervals are considered as candidates to the query q.
3. For each candidate object y, the distance $d(q,y)$ is calculated and if $d(q,y) \leq r$, then the object y is a solution to the query.

Implementation details are shown in algorithm 2. In this algorithm, S_{ij} represents the distance between the object y_i to the pivot p_j.

Figure 1 represents the data structure *spaghettis* in its original form. This structure is built using 4 pivots to index a database of 17 objects. The searching process is as follows. Assuming a query q, the distance to the pivots $\{8,7,4,6\}$, and a searching range $r = 2$, Figure 1 shows in dark gray the intervals $\{(6,10),(5,9),(2,6),(4,8)\}$ over which the searching is going to be carried out. Also, in this figure it is possible to see all the objects that belong to the intersection of all the intervals and then they are considered as candidates. Finally, the distance between the candidates and query has to be calculated in order to determine a solution from the candidates. The solution is given if the distance is lower than a searching range.

Algorithm 2 *Spaghettis*: Search Algorithm.

rangesearch(query q, range r)

1: {Let $\mathbb{Y} \subseteq \mathbb{X}$ be the database}
2: {Let P be set of pivots $p_1, \ldots, p_k \in \mathbb{X}$}
3: {Let D be the table of distances associated q}
4: {Let S be Spaghettis}
5: **for all** $p_i \in P$ **do**
6: $D_i \leftarrow d(q, p_i)$
7: **end for**
8: **for all** $y_i \in \mathbb{Y}$ **do**
9: $discarded \leftarrow false$
10: **for all** $p_j \in P$ **do**
11: **if** $D_j - r > S_{ij} \,||\, D_j + r < S_{ij}$ **then**
12: $discarded \leftarrow true$
13: break;
14: **end if**
15: **end for**
16: **if** !$discarded$ **then**
17: **if** $d(y_i, q) \leq r$ **then**
18: add to result
19: **end if**
20: **end if**
21: **end for**

4 GPU-based implementation

The main goal of this paper is to develop a GPU-based implementation of the range query algorithms.

This type of process intrinsically has a high data-level parallelism with a high computing requirements. For that reason, GPU computing is very useful in order to accelerate this process due to the fact that GPUs exploit in an efficient way data-level parallelism. Moreover, these devices provide the best cost-per-performance parallel architecture for implementing such algorithms.

This section is divided in two different parts. First, we show the exhaustive search GPU-based implementation, and next we present the spaghettis GPU-based implementation.

4.1 Exhaustive Search GPU-based Implementation

This implementation is an iterative process where in each iteration one kernel is executed, which calculates the distances between one particular query and every elements of the database. It is not possible to calculate all distances for all queries in only one kernel due to the GPU limitations (number of threads and memory capacity). In this kernel as many threads as number of elements in the database are launched. Each thread calculates the distance between one data of dataset and one particular query, and next, determines if this data is or not a valid solution.

4.2 Spaghettis GPU-based Implementation

In order to obtain better performance on GPU, we have made some changes on the original *Spaghettis* structure. We adapt the structure for that it is very similar to an array, which is more efficient in GPU computing. In this implementation, each row is associated with an object of dataset and each column to a pivot. Therefore, each cell contains the distance between the object and the pivot. Moreover, unlike the original version, the array is sorted by the first pivot. Thus, the cells of the same row is associated with the same object.

The parallelization of the searching algorithm has been splitted into three parts, which are the most computationally expensive parts of this algorithm. These parts correspond to the three steps presented in Subsection 3.2.

The first part consists in computing the distances between the set of queries, Q, and the set of pivots, P. In order to exploit the advantages of using a GPU platform is necessary a data structure which stores all distances. Therefore, this structure is implemented as a $Q \times P$ matrix which allows us to compute all distances at the same time in a single call to kernel. This part is implemented in one kernel with as many threads as number of queries. In fact, each thread solves independently the distance from a query to all pivots. The algorithm 3 shows a general pseudocode of this kernel.

Algorithm 3 Distance generator kernel

__global__ KDistances(queries Q, pivots P, distances D)

1: {Let D be the table of distances associated to q}
2: {Let i be thread Id }
3: **for all** $p_j \in P$ **do**
4: $\quad D_{ij} \leftarrow d(q_i, p_j)$
5: **end for**

The second part of the parallel implementation consists in determining if each element of the database is or not a candidate for every query. This part has been implemented as an iterative process. In each iteration the candidates for a particular query are computed in one kernel. As we have described above, it is not possible to calculate all candidates for every queries in only one kernel due to the GPU limitations. In this kernel as many threads as number of elements of the database are launched. Each thread of this kernel determines, for a given data (y_i) of the dataset, if this data is candidate or not. Thus, this kernel returns a list of candidates for a given query. Finally, when this process finishes we obtain one list of candidates for each query. This task is carried out by a kernel called *KCandidates* (see algorithm 4).

The kernel *KSolution* (see algorithm 5) correspond with the third part, and computes if each candidate is really a solution. In this kernel, the number of threads correponds to the number of candidates for each query. Each thread calculates the distance between one candidate and one query, and determines if

Algorithm 4 CUDA Search Algorithm.

__global__ KCandidates(range r, Spaghettis S, distances D, pivots P, candidates C)

```
1: {Let P be set of pivots p_1, ..., p_2 ∈ 𝕏}
2: {Let D be the table of distances associated q}
3: {Let C be list of candidates for q}
4: {Let i be thread Id }
5: discarded ← false
6: for all p_j ∈ P do
7:    if D_j − r > S_ij || D_j + r < S_ij then
8:       discarded ← true
9:       break;
10:   end if
11: end for
12: if !discarded then
13:    add to C (candidates)
14: end if
```

this candidate is or not solution. Finally, as result we obtain one list of solutions for each query.

In the three kernels, threads belonging to the same thread block operate over contiguous components of the arrays. Therefore, more efficient memory accesses are allowed. This is due to the abovementioned changes in the spaghettis structure.

Algorithm 5 CUDA final solutions for query q.

__global__ KSolution(range r, database $\mathbb{Y}$, candidates C, query q, solutions R)

```
1: {Let 𝕐 ⊆ 𝕏 be the database}
2: {Let C be list of candidates for q}
3: {Let R be list of solutions for q}
4: {Let i be thread Id }
5: if d(c_i, q) ≤ r then
6:    add to R (solutions)
7: end if
```

5 Experimental Evaluation

This section presents the experimental results obtained for the previous algorithms considering the Spanish dictionary as database. For this case study the generated *spaghettis* data structure is completely stored on the global memory of the GPU.

5.1 Experimental Environment

Tests made in one metric space from the Metric Spaces Library[5] were selected for this paper. This is a Spanish dictionary with 86,061 words, where the *edit distance* is used. This distance is defined as the minimum number of insertions, deletions or substitutions of characters needed to make one of the words equal to the other. We create the structure with the 90% of the dataset, and reserve the rest for queries. We have chosen this experimental environment because is the usual environment used to evaluate this type of algorithms.

Hardware platform used was a PC with the following main components:

- CPU: Intel Core 2 Quad at 2.66GHz and 4GB of main memory.
- GPU: GTX 285 with 240 cores and a main memory of 1 GB.

5.2 Experimental Results

The results presented in this section belong to a set of experiments with the following features:

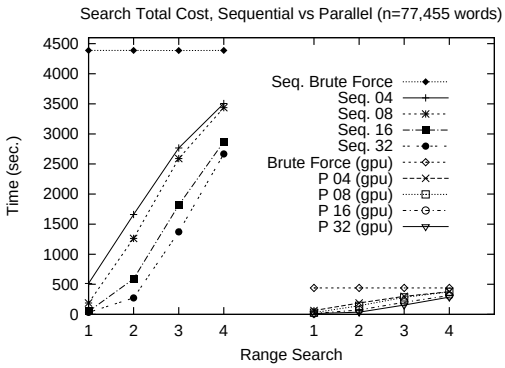
- The selection of pivots ware made randomly.
- The *spaghettis* structure was built considering 4, 8, 16, and 32 pivots.
- For each experiment, 8,606 queries were given over an *spaghettis* with 77,455 objects.
- For each query, a range search between 1 and 4 was considered.
- The execution time shown in this paper is the total time of all the processes for both versions, parallel and sequential. Therefore, in the case of parallel version, the execution time also includes the data transfer time between the main memory (CPU) and global device memory (GPU).

Figure 2(a) shows the execution time spent by the sequential and GPU implementation for *Spaghetttis* structure. Notice that the parallel version based on CUDA reduces dramatically the execution time, increasing the performance. Figure 2(b) shows in detail the time spent by the CUDA implementation. As reference, the execution time spent by the sequential and GPU implementation for the exhaustive search (Seq. and GPU Brute Force) is included in both figures (2(a) and 2(b)).

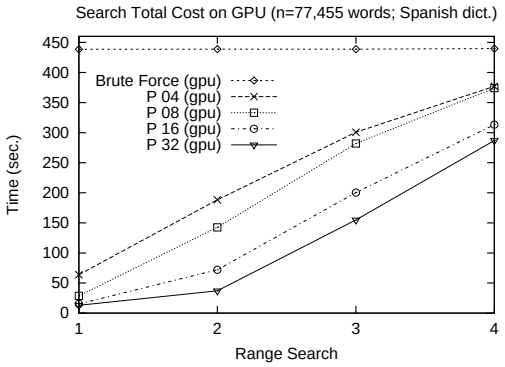
According to experimental results, it is interesting to discuss the following topics:

- As can be observed, the use of Spaghettis structure allows us to decrease the number of distance evaluations, due to that to compute the distance between all the database objects is avoided. In Figure 2 we can deduce that:
  - When the number of pivots increases the performance of search algorithm is much better in sequential and GPU versions.
  - The use of GPU decreases considerably the execution time in both versions, exhaustive search and emphSpaghettis structure.

[5] `www.sisap.org`.



(a) Sequential versus GPU results



(b) GPU details

Fig. 2. Comparative results of search costs for the space of words for *Spaghettis* metric structure (Spanish Dictionary). Number of pivots 4, 8, 16 and 32, and range search from 1 to 4.

– As can be observed in Figure 3 (range 1 and 2), the speed-up is smaller when the number of pivots is higher. Due to this fact, more number of pivots more workload for the threads. Moreover, when the range is higher (range 3 and 4) the speed-up increases, because the behaviour approaches to exhaustive search.

– There is an asymptotic speed-up around 9.5 (see Figure 3). It is possible to observe that this behaviour is shown when the range search is 4. But, in order to ensure this assertion, a proof considering a range search equal to 8 has been carried out.

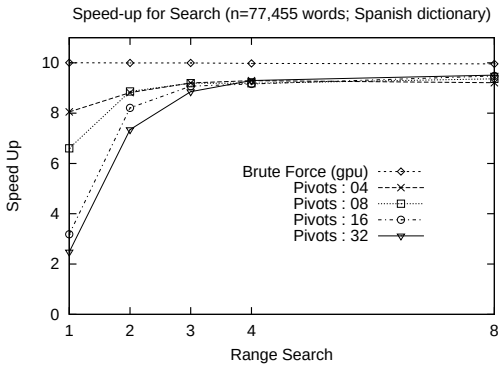


Fig. 3. Speed-up graphics to the space of words for *Spaghettis* metric structure (Spanish Dictionary).

6 Conclusions and Future Work

In this work, a parallel approach based on GPU has been carried out in order to reduce the execution time spent on the searching process of a query in a dataset using *Spaghettis* data structure.

This implementation has provided good results in terms of speed-up when considering suitable values for the input parameters as number of pivots and range search. In this case, a speed-up of 9.5 has been obtained.

To be able to continue with the study of this work in order to obtain more efficient implementations, and as future work, we have planned the following topics:

- To test the GPU-based implementation presented in this paper considering a different database.
- Moreover, we would like analyse the impact that different distance functions have on the global performance of this kind of algorithms, and on the acceleration obtained with parallel platforms. There are distance functions with a great computational load, like that presented in this paper, and others with minimum computational requirements. In these cases, hiding the overhead due to data transferences will be a challenge.
- In order to be able of executing the algorithms presented here on different GPU vendor platforms, OpenCL implementations will be carried out.
- To compare with other parallel platforms in terms of performance, energy consumption and economic cost. As a consequence, it is necessary to implement the work carried out here using MPI or OpenMP (or both) according to the target platform.

Acknowledgments

This work has been partially funded by research programs PR-F1-02IC-08, University of Magallanes, Chile and 29/C035-1, Academic Unit of Río Turbio, National University of Patagonia Austral, Argentina, and the Spanish project SATSIM (ref: CGL2010-20787-C02-02).

References

1. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, September 2001, pp. 33(3):273–321.
2. I. Kalantari and G. McDonald, "A data structure and an algorithm for the nearest point problem," *IEEE Transactions on Software Engineering*, vol. 9, no. 5, 1983.
3. J. Uhlmann, "Satisfying general proximity/similarity queries with metric trees." in *Information Processing Letters*, 1991, pp. 40:175–179.
4. P. Ciaccia, M. Patella, and P. Zezula, "M-tree : An efficient access method for similarity search in metric spaces." in *the 23st International Conference on VLDB*, 1997, pp. 426–435.
5. S. Brin, "Near neighbor search in large metric spaces." in *the 21st VLDB Conference*. Morgan Kaufmann Publishers, 1995, pp. 574–584.
6. R. Uribe and G. Navarro, "Egnat: A fully dynamic metric access method for secondary memory," in *Proc. 2nd International Workshop on Similarity Search and Applications (SISAP)*. IEEE CS Press, 2009, pp. 57–64.
7. G. Navarro, "Searching in metric spaces by spatial approximation," *The Very Large Databases Journal (VLDBJ)*, vol. 11, no. 1, pp. 28–46, 2002.
8. W. Gropp, E. Lusk, and A. Skelljum, *Using MPI:Portable Parallel Programming with the Message Passing Interface*, ser. Scientific and Engineering computation Series. Cambridge, MA: MIT Press, 1994.
9. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.
10. L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.
11. P. Zezula, P. Savino, F. Rabitti, G. Amato, and P. Ciaccia, "Processing m-trees with parallel resources," in *RIDE '98: Proceedings of the Workshop on Research Issues in Database Engineering*. Washington, DC, USA: IEEE Computer Society, 1998, p. 147.
12. A. Alpkocak, T. Danisman, and U. Tuba, "A parallel similarity search in high dimensional metric space using m-tree," in *Advanced Environments, Tools, and Applications for Cluster Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, vol. 2326, pp. 247–252.
13. M. Marín, R. Uribe, and R. Barrientos, "Searching and updating metric space databases using the parallel egnat," in *Computational Science - ICCS 2007*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, vol. 4487, pp. 229–236.
14. V. Gil-Costa, M. Marin, and N. Reyes, "Parallel query processing on distributed clustering indexes," *Journal of Discrete Algorithms*, vol. 7, no. 1, pp. 3–17, 2009.

15. V. Gil-Costa, R. Barrientos, M. Marin, and C. Bonacic, "Scheduling metric-space queries processing on multi-core processors," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 187–194, 2010.
16. E. Chávez, J. Marroquín, and R. Baeza-Yates, "Spaghettis: An array based algorithm for similarity queries in metric spaces," in *6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*. IEEE CS Press, 1999, pp. 38–46.
17. Wu-Feng and D. Manocha, "High-performance computing using accelerators," *Parallel Computing*, vol. 33, pp. 645–647, 2007.
18. "GPGPU. general-purpose computation using graphics hardware," http://www.gpgpu.org.
19. *NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, Version 2.3.* NVIDIA, 2009, http://developer.nvidia.com/object/gpucomputing.html.
20. Q. Kuang and L. Zhao, "A practical GPU based kNN algorithm," *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pp. 151–155, 2009.
21. V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," *Computer Vision and Pattern Recognition Workshop*, vol. 0, pp. 1–6, 2008.
22. B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Computational Science (ICCS)*, vol. 3994. Springer, 2006, pp. 196–199.
23. N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–10, 2009.
24. R. J. Barrientos, J. I. Gómez, C. Tenllado, and M. Prieto, "Heap based k-nearest neighbor search on GPUs," in *Congreso Español de Informática (CEDI))*, Valencia, Septiembre 2010.
25. L. Micó, J. Oncina, and E. Vidal, "A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements," *Pattern Recognition Letters*, vol. 15, pp. 9–17, 1994.